メタヒューリスティックアルゴリズム実装の手順と留意点

野々部　宏司

多くのメタヒューリスティクスは局所探索法を拡張したものと位置づけることができる．メタヒューリスティクスはアルゴリズム設計の自由度が高い柔軟な枠組みであり，解きたい最適化問題に対してまずは局所探索法のアルゴリズムを実装し，必要に応じてそれに改良を加え，メタヒューリスティックアルゴリズムに発展させていくことも可能である．本稿では，具体例としてジョブショップスケジューリング問題を取り上げ，局所探索法の実装から始め，これを反復局所探索法やタブー探索法に拡張していく流れについて紹介する．また，アルゴリズムを実装する際の留意点についても述べる．

キーワード：メタヒューリスティクス，局所探索法，アルゴリズム実装，ジョブショップスケジューリング

1. はじめに

メタヒューリスティクスは，最適化問題（特に組合せ最適化問題）に対する実用的解法として認知され，広く用いられている．メタヒューリスティクスの特徴として，アルゴリズムを実際に設計し，実装する際の自由度が高い点が挙げられる．比較的少ない手間でそれなりの性能を出すことができる一方で，問題構造をうまく利用することで高性能なアルゴリズムを作り上げることもできる．まずは単純な局所探索法から始め，必要に応じて改良を加えていくということも（結果的には二度手間となる可能性はあるが）可能である．特に，当初から超高性能アルゴリズムを開発することを目的とした学術研究のような場合ではなく，手元にある最適化問題をとりあえず解きたいという状況においては，アルゴリズムを実装しながら少しずつ改良を加えていくというアプローチは現実的な選択肢の1つと言えるであろう．本稿では，そのような状況を想定し，メタヒューリスティックアルゴリズムを実装していく手順の一例を具体例を用いて紹介する．また，実装の際の留意点についても述べる．

本稿では，例題としてジョブショップスケジューリング問題 (job-shop scheduling problem, JSP) を用いる．JSP は，代表的な汎用ソルバーである混合整数計画ソルバーが苦手とするタイプの問題であり，メタヒューリスティクスを試す価値の高い問題と考えられるからである．JSP は，n 個のジョブ $J_1, J_2, \ldots, J_n$ を m 台の機械 $M_1, M_2, \ldots, M_m$ で処理する際のスケジュールを求める問題である．ここで，各ジョブ J_i は m_i 個の作業 $O_{i,k}$ $(k = 1, 2, \ldots, m_i)$ から成り，これらの作業を $k = 1, 2, \ldots, m_i$ の順に処理することで J_i は完了する．また，各作業について，それを処理する機械と処理に要する時間は入力としてあらかじめ与えられる．このとき，すべてのジョブが完了する時刻（メイクスパン）が最小となるように各ジョブの各作業の処理開始時刻を決定することが JSP の目的である．ただし，機械は同時に複数の作業を処理することはできず，いったん処理を開始した作業を途中で中断することもできない．なお JSP では，すべてのジョブ J_i について，$m_i = m$ として m 個の作業はすべて異なる機械で処理される（J_i は各機械でちょうど1度ずつ処理される）ものと仮定することが多い．本稿でもそのように仮定する．

JSP は各作業の開始時刻を決定する問題ではあるが，各機械での作業の処理順序が決まれば，作業を1つずつ順に時間軸上に割りつけていくことで $\mathrm{O}(nm)$ 時間（作業数の線形時間）で開始時刻を定めることができる．そのため，JSP を各機械での処理順序（m 組の順列）を決定する問題ととらえることもできる．以下に例を示す．ジョブ数 $n = 4$，機械数 $m = 3$ とし，各作業 $O_{i,k}$ $(1 \le i \le 4,\ 1 \le k \le 3)$ について，それを処理する機械と処理時間が表 1 で与えられているとする．このとき，仮にすべての機械において $J_1 \to J_2 \to J_3 \to J_4$ の順に作業を処理するとすれば，例えば $O_{1,1}, O_{1,2}, O_{1,3}, O_{2,1}, O_{2,2}, \ldots$ の順に開始時刻

ののべ　こうじ
法政大学デザイン工学部
〒 102–8160 東京都千代田区富士見 2–17–1

表 1 ジョブショップスケジューリングの問題例

	$k=1$	$k=2$	$k=3$
J_1	M_1, 4	M_2, 2	M_3, 5
J_2	M_2, 2	M_3, 3	M_1, 6
J_3	M_3, 3	M_1, 4	M_2, 2
J_4	M_3, 6	M_2, 2	M_1, 3

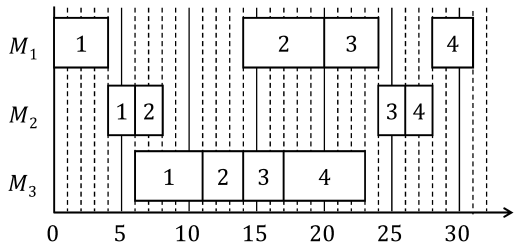

図 1 表 1 の問題例に対するスケジュール例

を決定していくことで，図 1 に示すメイクスパン 31 のスケジュールを得ることができる．なお，作業の処理順序の組合せによっては実行可能なスケジュールが存在しないことがあり[1]，実行可能性の判定は $O(nm)$ 時間でできる．

次節以降では，JSP に対する局所探索法の単純な実装から始め，近傍の改良を経て，反復局所探索法やタブー探索法に発展させていく流れについて述べる．なお，本稿の目的は JSP に対する高性能アルゴリズムを紹介することではないため，JSP に対するアルゴリズム自体に興味のある読者は文献 [2, 3] などを参照されたい．

2. 局所探索法の実装

2.1 基本設計

局所探索法を実装するには，その前の基本設計段階として，探索空間，解の評価関数，近傍を定義し，初期解の生成方法を定めておく必要がある．これらの決定はアルゴリズムの性能に大きな影響を与えるため慎重に行うべきであるが，ここではあまり凝った設計はせずに比較的単純な方法を採ることを考える．

探索空間については，前節で述べたとおり，JSP を各機械での処理順序を定める問題ととらえ，m 組の順列を解とし，その集合を探索空間とする．ただし，実行可能なスケジュールが存在しない解（実行不可能解）については，これらを探索空間に含めると解の評価がやや面倒になるため，ここでは除外することとする．よって解の評価関数は，単純にそこから得られるスケジュールのメイクスパン（目的関数）で定義する．解を 1 つ評価するのに $O(nm)$ 時間かかることになる．

現在の解から近傍解を生成する操作（近傍操作）については，順列に対する代表的な操作である挿入操作（順列内の 1 つの要素をほかの位置に挿入する操作）や交換操作（順列内の 2 つの要素の位置を交換する操作）が候補として考えられるであろう．まずは最も基本的な操作として，機械 M_j の処理順序を表す順列において h 番目の作業と $h+1$ 番目の作業を交換する操作 $swap(j,h)$ を考え，現在の解に $swap(j,h)$ $(1 \leq j \leq m,\ 1 \leq h \leq n-1)$ を適用することで得られる解の集合を近傍としよう．ただし，$swap(j,h)$ によって生成された解が実行不可能である場合は，近傍解としては扱わない（移動先の解候補としない）こととする．近傍の大きさは $O(nm)$ である．

初期解はランダムに生成することとする．ただし，単にランダムな順列を m 個生成して組み合わせるだけでは実行不可能解となる可能性があるため，ここでは，各機械での処理順序が何も決まっていない状態から始め，割りつけ可能な作業が複数存在する場合にはランダムに 1 つ選びながら作業を 1 つずつ時間軸上に割りつけていくことで処理順序を決定するものとする．この計算は $O(nm)$ 時間で可能である．

2.2 実装手順

それでは実際にアルゴリズムを実装してみよう．ここでは以下の手順に従って実装を行う．

(i) 入力データの読み込みを行うプログラムを作成する．
(ii) 解の生成，解の評価，解の出力を行うルーチンをそれぞれ追加する．
(iii) 近傍探索を行うルーチンを追加する．
(iv) 初期解を変えて局所探索を繰り返し実行するルーチンを追加する．

(i) について，入力データ（問題例）のフォーマットを定めておく必要がある．JSP の場合はベンチマーク問題（[1] など）が存在するので，ここではそのフォーマットに従うこととするが，入力フォーマットを自由に設定できる場合には，数値の羅列などではなく可読性の高いフォーマットを採用することを検討してもよい．字句解析や構文解析が必要となりデータの読み込みにかかる時間が増す可能性もあるが，プログラムの

[1] 例えば表 1 の例で，M_1 では $J_2 \to J_1$ の順に，M_2 では $J_1 \to J_2$ の順にそれぞれ処理するものとすると，$O_{2,3}$ は $O_{1,1}$ よりも先に，$O_{1,2}$ は $O_{2,1}$ よりも先にそれぞれ処理することになる．一方，問題の定義より，$O_{1,1}$ は $O_{1,2}$ よりも先に，$O_{2,1}$ は $O_{2,3}$ よりも先にそれぞれ処理しなくてはならず，これらすべてを同時に満たすことは不可能である．

利便性は高まる．

(ii) では，ランダムな解を 1 つ生成し，それに従って各作業の開始時刻とメイクスパンを計算できるようにする．解の出力について，解は m 組の順列で表されるが，ここではスケジュール自体を確認できるように，各作業の開始時刻をメイクスパンとともにファイルに出力できるようにする．さらに，逆にファイルから各作業の開始時刻を読み込んで，スケジュールの実行可能性やメイクスパンの値のチェックができるようにしておく．一般に，プログラムが正しく動作していることを確認するためにも，出力解がすべての制約を満たすことや目的関数値が正しいことを確認できる仕組みを整えておくべきであるが，この時点でこれを用意しておけば，この後のデバッグ作業にも役立つ．

(iii) は局所探索法の核となる処理の実装である．現在の解に近傍操作を適用して近傍解を生成し，その評価を行うという処理を繰り返すことが中心となる．このとき，近傍解をどのような順序で調べ，どのタイミングでどの改善解に移動するかの戦略，すなわち移動戦略を決める必要がある．代表的な戦略として，即時移動戦略（近傍解を順番に調べていき，改善解が見つかった時点でただちに移動）と最良移動戦略（近傍解をすべて調べ，最も評価値の高い近傍解に移動）が挙げられるが，どちらが良い性能を与えるかは一概には言えない．一般に，即時移動戦略のほうが解の移動 1 回当たりの計算時間が少なく，より有効であることが多いとされているが，問題や近傍の大きさなどにもよるため，可能であれば両方を試してみるのがよい．

即時移動戦略の場合，近傍解を調べる順序にも注意すべきである．通常，近傍解およびそれを生成する近傍操作は，1 つあるいは複数のインデックス（例えば $swap(j,h)$ の場合は j と h の 2 つ）で特定され，for ループなどの繰り返し処理によって近傍の探索が行われる．このとき，近傍解をインデックスの順に調べていく方法が単純で実装も容易であるが，この場合，改善解が見つかった時点で解の移動を行った後，次回の反復でも再びインデックスの最初から近傍解を調べていくことになる．そのため調べる順番が早いインデックスで特定される近傍解ほど移動先の解として選ばれやすく，探索に偏りが生じることになる．また，改善解が見つかる前に調べた，改善をもたらさない近傍操作を次回もまた最初から順に調べていくことになるため，それらのほとんどが解の移動後も改善をもたらさないままであれば，結果として無駄な計算を繰り返し行うことになる．探索順序が計算結果に与える影響を事前に予測することは容易ではないが，以下の点に留意するとよいであろう．

まず，インデックスが単なる識別番号であって値の大小が特に意味を持たない場合を考えよう．例えば JSP に対する $swap(j,h)$ の j がこれに対応する．j は機械の識別番号であって，番号をつけ替えても問題は本質的には変わらないからである．アルゴリズムの挙動が問題の本質的ではない情報に左右されることは好ましいことではないと考えれば，このような場合，特定のインデックスを優先的に調べる（あえて探索に偏りを持たせる）理由が特にない限り，探索順序はランダムにするのがよいと思われる．具体的には以下のようにすればよい．ただし，ここでは近傍解の個数が N で，それぞれの解が $0, 1, \ldots, N-1$ のインデックスによって特定されるものとしている．また `idx[]` は，要素として $0, 1, \ldots, N-1$ をそれぞれ 1 つずつ含む大きさ N の配列であるとし，`randbetween(`a`,`b`)` は整数 a, b に対して a 以上 b 以下の整数値をランダムに返す関数であるとする．

```
do{
  for(p=0; p<N; ++p){
    q=randbetween(p,N-1);
    c=idx[q]; idx[q]=idx[p]; idx[p]=c;
    if(c で特定される解 x' が x よりも良い){
      x を x' で置き換えて break;
     }
   }
} while(p<N);
```

ここで，`for` ループが終了した時点で `p<N` が成り立つことは，改善解が見つかって解の移動が行われたことを意味する．また，配列 `idx[]` にインデックスがあらかじめランダムに並べられていれば，上のプログラムの 3，4 行目を

```
  q=(q+1)%N; c=idx[q];
```

で置き換えることで，探索順序を毎回ランダムに変えることなく，問題の本質的ではない情報が探索に影響を及ぼすことを避けることができる．ここで，変数 `q` は `do` ループの前に初期化されているものとし，`for` ループの開始時には初期化しない．このようにすることで，改善解に移動した後，次回の探索はその次のインデックスから始めることになるため，前回の反復で改善をもたらさなかった近傍操作を再度順番に調べる

ことも避けられる．

次に，インデックスの大小に何らかの意味がある場合を考えよう．例えば，$swap(j,h)$ の h がこれに対応する．h の値が小さい（大きい）ほど，スケジュールの前（後ろ）のほうの箇所を変更することになり，スケジュール全体に及ぼす影響は大きく（小さく）なる傾向があると言える．このような場合は，近傍解をインデックスの昇順または降順で調べ，あえて探索に偏りを持たせることで性能を向上させることができる可能性がある（この場合は，毎回インデックスの最初から探索を始めるのが自然であると思われる）．もちろん，逆に性能を悪化させる危険性もあるため，ランダム順とすることも含めて，どの戦略を採用するかは実際に試してみて判断するのがよいであろう．

最良移動戦略を用いるときでも，近傍内に評価値の同じ解が複数存在する場合には探索順序が性能に影響を及ぼす可能性がある．この場合も即時移動戦略のときと同様に考えればよい．なお，探索の高速化のためにデータ構造を工夫しているなど，探索順序を変更することが難しい場合がある．その場合でも，近傍探索中，最良の評価値を持つ近傍解はすべて保持するようにしておき，最後にその中からランダムに 1 つを選択することにすれば，ランダムな順序で探索することと同じ効果が期待できる．

近傍操作として $swap(j,h)$ を用いた今回の例では，即時移動戦略と最良移動戦略の両方を実装した．また，インデックス j についてはランダム順として，インデックス h については昇順・降順・ランダム順の 3 つを実装した．なお，近傍探索は j と h の 2 重 for ループで実現し，h を外側のループ，j を内側のループとしている．

以上で局所探索法のプログラムが完成したことになる．今回は，この段階で局所探索 1 回当たりの計算時間が非常に短いことが確認できたので，(iv) の手順として多スタート局所探索法 (multi-start local search, MLS) のプログラムを作成し，これを用いて計算実験を行うこととする．なお，MLS のプログラムは計算時間が上限に達したときに終了するものとし，上限の値は実行時に指定できるようにしている．

2.3 計算実験

プログラムが正常に動くことを確認したうえで計算実験を行い，移動戦略の決定や性能（得られる解の精度や計算時間）の確認を行う．今回の例では，多くの論文で使用されているベンチマーク問題例の中から ft10, la40, abz7 と呼ばれる 3 問を選び，実験用のデータとして用いることとした．これらの問題例の規模と最適値は表 2 のとおりである．

表 2 ベンチマーク問題例

問題例	ジョブ数 n	機械数 m	最適値
ft10	10	10	930
la40	15	15	1222
abz7	20	15	657

表 3 MLS における移動戦略・探索順序による影響

問題例	即時移動戦略			最良移動戦略		
	昇順	降順	ランダム	昇順	降順	ランダム
ft10	1022.0	**1018.5**	1019.0	1036.6	1032.4	1034.6
	29115	31625	56427	19153	19113	19249
la40	1443.0	**1432.0**	1449.0	1492.8	1493.6	1505.2
	3221	3601	9088	2499	2505	2520
abz7	831.0	**824.3**	832.6	863.6	865.5	870.9
	1803	2101	6212	1629	1598	1601

移動戦略とインデックス h の走査順が異なる 6 種類のアルゴリズムについて，乱数の種を変えて 10 回ずつプログラムを実行した結果の概要を表 3 に示す．各アルゴリズムについて，最良解のメイクスパンの平均値を上段に，計算時間内に実行した局所探索の回数（初期解を生成した回数）の平均値を下段にそれぞれ記載している．なお，1 回当たりの計算時間はすべて 60 秒としている[2]．即時移動戦略のほうが最良移動戦略よりも性能が良く，インデックス h については，即時移動戦略の場合，降順で近傍解を調べるほうが昇順やランダム順で調べるよりもやや性能が良いことが確認できる．また，局所探索の実行回数が多いほど良い結果が得られるとは限らないことがわかる．今回の例では，計算結果から「即時移動戦略・降順」の組合せを採用することとし，このアルゴリズムの 10 回の計算結果（メイクスパンの最小値・平均値・最大値）を，表 4〜6 の「MLS（近傍改良前）」の行に示す．

なお，表 3 の結果からは探索順序が性能に与える影響はさほど大きくはないととらえることもできる．局所探索法を実装する際，余裕があれば探索順序についても十分検討するほうが望ましいが，高い性能が強く求められる状況でなければ，ランダム順だけを試すということも考えられるであろう．

[2] 使用した計算機の CPU は Core i7-3520 M 2.90 Ghz, RAM は 8GB，OS は Windows 8 64 ビットである．

表 4　ft10（最適値 930）に対する計算結果

アルゴリズム	最小値	平均値	最大値
MLS（近傍改良前）	985	1018.5	1030
MLS（近傍改良後）	959	977.1	987
ILS（近傍改良前）	937	976.7	1016
ILS（近傍改良後）	936	952.0	967
TS	930	930.0	930

表 5　la40（最適値 1222）に対する計算結果

アルゴリズム	最小値	平均値	最大値
MLS（近傍改良前）	1398	1432.0	1452
MLS（近傍改良後）	1299	1343.1	1377
ILS（近傍改良前）	1266	1324.8	1431
ILS（近傍改良後）	1239	1253.5	1266
TS	1228	1232.5	1234

表 6　abz7（最適値 657）に対する計算結果

アルゴリズム	最小値	平均値	最大値
MLS（近傍改良前）	807	824.3	838
MLS（近傍改良後）	767	776.8	782
ILS（近傍改良前）	726	752.6	776
ILS（近傍改良後）	693	714.5	730
TS	668	674.3	678

3. 局所探索法の改良・拡張

前節で作成したプログラムでは解の精度が十分ではないと判断したとして，これを改良することを考えよう．改良の方法としては，(i) 近傍解の評価値の計算の高速化，(ii) 近傍の改良，(iii) メタヒューリスティクスへの拡張などが考えられる．局所探索法やメタヒューリスティクスにおいて，近傍解の評価値の計算は実行回数が多いため，これを高速化できればプログラム全体の高速化につながる．とくにオーダー評価の意味での改善が実現できれば，その効果は非常に大きなものとなる．しかし高速化は，探索性能を本質的に向上させるものではないため，多くの場合，(ii) や (iii)，その他探索性能を向上させるための手法と併用することが望ましい[3]．本稿では，(i) については割愛することとし，(ii) と (iii) について述べる．(iii) では，反復局所探索法 (iterated local search，ILS) とタブー探索法 (tabu search, TS) を扱う．ILS や TS の概要については，本特集の別記事や文献 [4] などを参照されたい．

3.1 近傍の改良

局所探索法では，一般に近傍は大きいほど得られる解の精度は高まる．ここでは JSP の例において，$swap(j,h)$ を拡張した近傍操作として，機械 M_j の処理順序を表す順列において h 番目の作業（以下，移動作業と呼ぶ）を $h-d$ 番目の作業の前に挿入する操作 $shift(j,h,d)$ を考えることにしよう．ここで，$1 \leq j \leq m$，$2 \leq h \leq n$，$1 \leq d \leq h-1$ であり，$shift(j,h,1)$ は $swap(j,h-1)$ と同じ操作である．ただし，(j,h,d) のすべての組合せについて $shift(j,h,d)$ を適用するのではなく，無駄な計算を省くため以下の 2 つの制限を行う．

1. 移動作業をクリティカルパス上の作業（その処理開始時刻が少しでも遅れると，処理順序を変えたり制約に違反したりしない限りメイクスパンが増加してしまう作業）に制限する．クリティカルパス上にない作業の処理順序を早めてもメイクスパンは減少しないからである．図 1 の例では，M_1 で処理するジョブ 2，3，4 の作業，M_2 で処理するジョブ 3，4 の作業，M_3 で処理するジョブ 2 の作業が対象となる．
2. 機械 M_j で h 番目に処理する作業（移動作業）を $O_{i,k}$，$h-d$ 番目に処理する作業を $O_{i',k'}$ として，$O_{i',k'}$ の完了時刻が $O_{i,k-1}$ の完了時刻（$k=1$ の場合は 0）よりも大きい場合に制限する．$O_{i,k}$ は $O_{i,k-1}$ の完了時刻以降にしか開始することはできないため，$O_{i',k'}$ の完了時刻がこれよりも大きくない限り，$O_{i,k}$ の処理順序を $O_{i',k'}$ の前に移動する意味はない（$O_{i',k'}$ の後に処理するとしても $O_{i,k}$ の開始時刻は変わらない）からである．また，現在の解に $shift(j,h,d)$ を適用することで解が実行不可能になることがあるが，それは処理順序の関係から $O_{i,k-1}$ が $O_{i',k'}$ よりも後に処理することになっている場合であり，上のように制限することでこれを防ぐことができる[4]．図 1 の例では，1 つ目の制限と合わせて $(j,h,d)=(1,3,1),(2,4,1),(3,2,1)$ だけが対象となる．

[3] ただし，複雑あるいは大規模な問題を扱うなど，単純な局所探索法の実行にすら多くの計算時間がかかる場合には，高速化自体に大きな意味がある．

[4] 例えば図 1 の解に $shift(1,3,2)$ を適用すると実行不可能になる．これは，移動作業（M_1 で 3 番目に処理する作業）$O_{3,2}$ の前に処理すべきジョブ 3 の作業 $O_{3,1}$ が，M_1 で最初に処理する作業 $O_{1,1}$ よりも後に処理する（$O_{1,1} \to O_{1,2} \to O_{1,3} \to O_{2,2} \to O_{3,1}$ の順に処理する）ことになっているからである．このとき，$O_{1,1}$ の完了時刻 4 は $O_{3,1}$ の完了時刻 17 以下であり，確かに $shift(1,3,2)$ は適用対象外となっている．

以上のように制限した(j,h,d)は，その総数をNとして，$O(nm+N)$時間で列挙することができる．

MLSのプログラムにおいて近傍をこのように変更し，前章と同様に計算実験を行った結果を表4〜6の「MLS（近傍改良後）」の行に示す．この結果から，近傍を改良した効果を確認することができる．

3.2 反復局所探索法への拡張

今回の例のようにMLSのプログラムがすでに完成していれば，それをILSに発展させることは実装上はそれほど困難ではない．MLSでは毎回ランダムに初期解を生成していたところを，例えば，それまでに得られている最良解（暫定解）にランダムな摂動を加えて新たな解を生成するように変更すればよい[5]．今回のJSPの例では，ランダム摂動として$swap(j,h_1,h_2)$をランダムに複数回適用することとしよう．ここで，$swap(j,h_1,h_2)$は機械jの処理順序を表す順列においてh_1番目の作業とh_2番目の作業を交換する操作であり，$1\leq j\leq m$，$1\leq h_1<h_2\leq n$，$h_2-h_1\geq 2$とする．$h_2-h_1\geq 2$を条件としているのは，局所探索で用いる近傍操作には含まれない操作のみをランダム摂動に用いるためである．なお，$swap(j,h_1,h_2)$を適用することで解が実行不可能になってしまう場合には，解を適用前に戻し，再度ランダムに(j,h_1,h_2)を生成して操作の適用を試みる．この場合，取り消した操作は適用回数にはカウントしない．

$swap(j,h_1,h_2)$の適用回数は，少なければ次に実行する局所探索において暫定解の近くをより入念に探索することになり，逆に多ければ暫定解から離れた領域で探索を行うことになるため，探索の集中化と多様化のバランスを決める役割を持つものであると言える．集中化と多様化を両立させるために，例えば初めは比較的小さな値に設定しておき，暫定解が更新されない場合に徐々に大きくしていくなど，計算状況に応じて適用回数を適応的に変化させる方法が有効であると考えられるが，ここでは単純に，新たな初期解を生成しようとする度に1以上U以下の整数値uをランダムに生成し，$swap(j,h_1,h_2)$をu回適用することとする．ここで，適用回数上限Uはパラメータであり，計算実験を通して適切な値を探ることとする．今回の適用回数のように値を動的に変化させるほうがよさそうなパラメータについて，とりあえず一定範囲内でランダムに変化させてみるのも1つの方法であろう．Uのような別のパラメータが必要にはなるが，それが計算結果に与える影響はランダム性を導入している分，元のパラメータよりも小さく，パラメータの値の調整が容易になることが期待できる．

2節のMLSを拡張したILSと近傍を改良した3.1項のMLSを拡張したILSの計算結果を，表4〜6の「ILS（近傍改良前）」と「ILS（近傍改良後）」の行にそれぞれ示す．なお，Uの値は10, 20, 30のそれぞれを試した結果から，すべての問題例について$U=20$としている[6]．計算結果から，MLSをILSに拡張することで性能が大きく向上していることがわかる．また，近傍を改良したMLSよりも，改良前の単純な近傍を用いたILSのほうが良い結果が得られていることから，近傍の適当な改良方法が見つからない場合でも，とりあえずILSの手法を試す価値はあると言えよう．

3.3 タブー探索法への拡張

タブー探索法は，最良移動戦略を用いた局所探索法のプログラムを基に改良することで実装することができる．タブー探索法に組み込む手法やアイデアとしてはさまざまなものがあるが，ここではとりあえずタブーリストを追加することだけを考える．なお，タブー探索法では近傍解（タブーリストによって移動が禁止されているものを除く）すべてを探索の対象とするのではなく，何らかの方法で対象を見込みのある近傍解に限定する戦略（候補リスト戦略と呼ばれる）がよく用いられ，これが探索性能を決定づけることも多い．JSPの例では，3.1項の$shift(j,h,d)$を用いた近傍がこの戦略に沿ったものであるといえ，ここではこの近傍を用いることとする．

タブーリストの実現方法としては，解の移動の際に実際に適用した近傍操作の移動作業$O_{i,k}$について，ある反復期間の間，再度$O_{i,k}$を移動作業とする近傍操作は行わないこととする．具体的には，各要素$T[i,k]$が各作業$O_{i,k}$に対応する2次元配列Tを$T[i,k]=0$$(\forall i,\forall k)$で初期化して用意しておき，$r$回目の近傍探索において，$T[i,k]\geq r$である$O_{i,k}$を移動作業とする近傍操作は探索対象から除外することとする．そして，r回目の解の移動を行った際には，その移動を実現する近傍操作の移動作業$O_{i,k}$に対応するTの要素を$T[i,k]=r+\ell$によって更新する．ここで，ℓはタブー期間と呼ばれるパラメータであり，3.2項における$swap(j,h_1,h_2)$の適用回数と同様，探索状況に応じて適応的に変化させることが望ましいと考えられる．ここでも上限を表すパラメータLを導入し，解の移動の

[5] 初期解生成の方法としてほかにもいろいろなアイデアが考えられるが，ここでは立ち入らない．

[6] 詳細な結果は省略するが，Uの値が探索性能に与える影響はそれほど大きなものではなかった．

度に 1 以上 L 以下の整数値をランダムに生成し，その値を ℓ として用いることとする．なお，タブーリストによって移動が禁止されている近傍解であっても，それが過去の探索で得られている最良解よりも良い解（メイクスパンが小さい解）であれば，移動先の解として認めることとする．また，探索対象のすべての近傍操作がタブーリストによって禁止されている場合は，解の移動を行わずに次の反復に移るものとする（反復回数 r が 1 増加するため，いずれ禁止期間が終了する近傍操作が出てくる）[7]．

計算結果を表 4〜6 の「TS」の行に示す．タブー期間の上限 L については，いくつかの値で計算を行った結果から，ジョブ数 n を用いて $L = 2n$ とすることとしている．今回実装したプログラムはタブー探索法としては比較的単純な実装ではあるが，ILS よりもさらに性能が向上し，特に ft10 については 10 回すべての実行で最適値が得られるまでに改良されていることがわかる．今回用いた 3.1 項の近傍がタブー探索法に適していたということが言えるであろう．

4. 実装の際の留意点

最後に，メタヒューリスティックアルゴリズムを実装する際の留意点について述べる．

プログラムの挙動の確認や性能の評価のために，可能な限り計算結果を再現できるようにしておくことが望ましい．メタヒューリスティックアルゴリズムの場合，計算過程に乱数を用いることが多いが，乱数の種として実行時の時刻を用いたり，コンパイラや計算機環境によって挙動が変わるような組込み関数を乱数生成に利用したりすることは原則として避けるほうがよい．また，アルゴリズムの挙動を計算途中で切り替える場合（例えば複数の探索ルールを順番に試す場合など），切り替えのタイミングの決定に CPU 時間を用いることは，再現性の観点からは好ましくない．

また，メタヒューリスティクスは通常複数のパラメータをもつ．これらの設定値は，後から確認できるよう実行時に出力するようにし，ログファイルに記録しておくことを推奨する． なお，パラメータには実行時にオプションによって与えるものとソースコードに埋め込むものがあるが，ユーザが指定・変更する可能性のあるものはオプションで与えられるようにしておくのがよいであろう．また，ソースコードに埋め込む場合には，`const` や `enum` を用いるなどして，具体的な設定値はソースコードの 1 カ所だけに記述するようにすることが望ましい．

メタヒューリスティックアルゴリズムは，開発者が意図したとおりの近傍探索が行われていなかったとしても，解の置き換えが正しく行われていれば出力解自体には問題がないことが多く，バグの見落としにつながる危険性がある．そのためプログラムを作成した後，計算結果だけでなくアルゴリズムの挙動（計算過程）を細かく確認することが必要であると言える．バグがないとしても，探索性能を向上させるためにはやはり計算過程をよく眺め，探索の傾向を十分把握することが重要となる．例えば目的関数値の推移から，解の変化が少なく探索の多様化が十分ではないなど，改善すべき点が見つかることもある．

5. おわりに

本稿では，メタヒューリスティックアルゴリズムの実装手順の一例を紹介するとともに，実装時の留意点について述べた．考えられる手順や方法はほかにも多数存在するため，本稿の内容は 1 つの事例としてとらえ，実装の際の参考にしていただければ幸いである．

参考文献

[1] J. E. Beasley, OR-Library, Job shop scheduling, URL: `http://people.brunel.ac.uk/~mastjjb/jeb/orlib/jobshopinfo.html`, accessed on August 31, 2013.

[2] J. Błażewicz, W. Domschke, E. Pesch, The job shop scheduling problem: Conventional and new solution techniques, *European Journal of Operational Research*, **93**, 1–33, 1996.

[3] J. Błażewicz, K.H. Ecker, E. Pesch, G. Schmidt and J. Weglarz, *Handbook on Scheduling: From Theory to Applications*, Springer, 2007.

[4] 柳浦睦憲，茨木俊秀，『組合せ最適化—メタ戦略を中心として—』，朝倉書店，2001.

[7] 反復回数 r を 1 ずつ増加させるのではなく，探索対象となっている近傍操作の $T[i,k]$ の値の最小値を T' として r を $T'+1$ までスキップさせれば，直後の反復において解の移動を行うことができるようになる．

